Notas de Programación Orientada a Objetos. ver. 1.0 *

Carlos Alberto Fernández y Fernández.
Instituto de Electrónica y computación.
UNIVERSIDAD TECNOLÓGICA DE LA MIXTECA.

Contenido.

Características de C++.

Algunas características de C++ que no tienen que ver directamente con la programación orientada a objetos.

Comentarios en C++.

Los comentarios en C son:

/* */

En C++ los comentarios pueden ser además de una sola línea:

```
// este es un comentario en C++
```

Acabando el comentario el final de la línea, lo que quiere decir que el programador no se preocupa por cerrar el comentario.

Flujo de entrada/salida

En C, la salida y entrada estándar estaba dada, entre por printf y scanf principalmente o funciones similares, para el manejo de los tipos da datos simples y las cadenas. En C++ se proporcionan a través de la librería iostream.h, el cual debe ser insertado a través de un #include. Las insturcciones son:

- cout Utiliza el flujo salida estándar
Que se apoya del operador <<, el cual se conoce como *operador de inserción de flujo* "colocar en"

- cin Utiliza el flujo de entrada estándar.
Que se apoya del operador >>, conocido como *operador de extracción de flujo* "obtener de"

Los operadores de inserción y de extracción de flujo no requieren cadenas de formato (%s, %f), ni especificadores de tipo. C++ reconoce de manera automática que tipos de datos son extraídos o introducidos.

En el caso de el operador de extracción (>>) no se requiere el operador de dirección &.

De tal forma un código de desplegado con printf y scanf de la forma:

```
Printf("Número: ");
Scanf("%d", &num);
Printf("El valor leído es: " %d\n", num);
```

Sería en C++ de la siguiente manera:

```
Cout << "Número";
Cin >> num;
Cout << "El valor leído es: " << num << '\n';
```

Funciones en línea.

Las funciones en línea, se refiere a introducir un calificador *inline* a una función de manera que le sugiera al compilador que genere una copia del código de la función en lugar de la llamada.

Ayuda a reducir el número de llamadas a funciones reduciendo el tiempo de ejecución en algunos casos, pero en contraparte puede aumentar el tamaño del programa.

A diferencia de las macros, las funciones inline si incluyen verificación de tipos y son reconocidas por el depurador.

Las funciones inline deben usarse sólo para funciones chicas que se usen frecuentemente.

El compilador desecha las solicitudes inline para programas que incluyan un ciclo, un switch o un goto. Tampoco si no tienen return (aunque no regresen valores) o si contienen variables de tipo *static*. Y lógicamente no genera una función inline para funciones recursiovas.

Declaración:

Inline <declaración de la función>

Ejemplo:

```
Inline float suma (float a, float b) {
        Return a+b;
}

inline int max( int a, int b) {
        return (a > b) ? a :  b;
}
```

Declaraciones de variables.

Mientras que en C, las declaraciones deben ir en la función antes de cualquier línea ejecutable, en C++ pueden ser introducidas en cualquier punto, con la condición de que la declaración esté antes de la utilización de lo declarado.

También puede declararse una variable en la sección de inicialización de la instrucción for, pero es incorrecto declarar una variable en la expresión condicional del while, do-while, for, if o switch.

El alcance de las variables en C++ es por bloques. Una variable es vista a partir de su declaración y hasta la llave } del nivel en que se declaró. Lo cual quiere decir que las instrucciones anteriores a su declaración no pueden hacer uso de la variable ni después de finalizado el bloque.

Operador de resolución de alcance.

Se puede utilizar el operador de resolución de alcance :: se refiere a una variable, con un alcance de archivo (variable global).

Esto le permite al identificador ser visible aún si el identificador se encuentra oculto.
Ejemplo:

```
Float h;

Void g(int h) {
        Float a;
        Int b;

        A=::h;    // a se inicializa con la variable global h

        B=h;      // b se inicializa con la variable local h
}
```

Valores por Default.

Las funciones en C++ pueden tener valores por default. Estos valores son los que toman los parámetros en caso de que en una llamada a la función no se encuentren especificados.

Los valores por omisión deben encontrarse en los parámetros que estén más a la derecha. Del mismo modo en la llamada se deben empezar a omitir los valores de la extrema derecha.

C++ no permite la llamada omitiendo un valor antes de la extrema derecha de los argumentos:

```
punto( , 8);
```

Parámetros por referencia.

En C todos los pasos de parámetros son por valor, aunque se pueden enviar parámetros "por referencia" al enviar por valor la dirección de un dato (variable, estructura, objeto), de manera que se pueda accesar directamente el área de memoria del dato del que se recibió su dirección.

C++ introduce parámetros por referencia reales. La manera en que se definen es agregando el símbolo & de la misma manera que se coloca el *: después del tipo de dato en el prototipo y en la declaración de la función.

Notar que no hay diferencia en el manejo de un parámetro por referencia y uno por valor, lo que puede ocasionar ciertos errores de programación.

Variables de referencia.

También puede declararse una variable por referencia que puede ser utilizada como un seudónimo o alias. Ejemplo:

```
Int max=1000, &smax=max;        //declaro max y  smax es un alias de max

Smax++;                         //incremento en uno max a través de su alias.
```

Esta declaración no reserva espacio para el dato, pues es como un apuntador pero se maneja como una variable normal.

Supuestamente no se permite reasignar la variable por referencia a otra variable, pero borlandC si lo permite.

Asignación de memoria en C++ (new y delete)

En el ANSI C, se utilizan malloc y free para asignar y liberar dinámicamente memoria:

```
float *f;
f=(float *) malloc(sizeof(float));
----
free(f);
```

Se debe indicar el tamaño a través de sizeof y utilizar una máscara (cast) para designar el tipo apropiado de dato.

En C++, existen dos operadores para asignación y liberación de memoria dinámica: new y delete.

```
float *f;
f= new float;
----
delete f;
```

El operador new crea automáticamente un área de memoria del tamaño adecuado. Si no se pudo asignar le memoria se regresa un apuntador nulo (NULL ó 0). Nótese que en C++ se trata de operadores que forman parte del lenguaje, no de funciones de librería.

El operador delete libera la memoria asignada previamente por new. No se debe tratar de liberar memoria ya liberada o no asignada con new.

Es posible hacer asignaciones de memoria con inicialización:

```
Int *max= new int (1000);
```

También es posible crear arreglos dinámicamente.

```
Char *cad;
Cad= new char [30];
---
delete [] cad;
```

Usar delete sin los corchetes puede no liberar adecuadamente la memoria, sobre todo si son elementos de un tipo definido por el usuario.

Plantillas o "templates"

Cuando las operaciones son idénticas pero requieren de diferentes tipos de datos, podemos usar lo que se conoce como templates o plantillas de función.

El termino de plantilla es porque el código sirve como base (o plantilla) a diferentes tipos de datos. C++ genera al compilar el código objeto de las funciones para cada tipo de dato involucrado en las llamadas.

Las definiciones de plantilla se escriben con la palabra clave template, con una lista de parámetros formales entre < >. Cada parámetro formal lleva la palabra clave class.

Cada parámetro formal puede ser usado para sustituir a: tipos de datos básicos, estructurados o definidos por el usuario, tipos de los argumentos, tipo de regreso de la función y para variables dentro de la función.

Introducción a la programación orientada a objetos. [6][7]

Programación no estructurada.

Comúnmente, las personas empiezan a aprender a programar escribiendo programas pequeños y sencillos consistentes en un solo programa principal. Aquí "programa principal" se refiere a una secuencia de comandos o instrucciones que modifican datos que son a su vez globales en el transcurso de todo el programa.

Programación procedural.

Con la programación procedural se pueden combinar las secuencias de instrucciones repetitivas en un solo lugar.

Una llamada de procedimiento se utiliza para invocar al procedimiento.
Después de que la secuencia es procesada, el flujo de control procede exactamente después de la posición donde la llamada fue hecha

Al introducir parámetros así como procedimientos de procedimientos (subprocedimientos) los programas ahora pueden ser escritos en forma más estructurada y con menos errores.

Por ejemplo, si un procedimiento ya es correcto, cada vez que es usado produce resultados correctos. Por consecuencia, en caso de errores, se puede reducir la búsqueda a aquellos lugares que todavía no han sido revisados.

De este modo, un programa puede ser visto como una secuencia de llamadas a procedimientos . El programa principal es responsable de pasar los datos a las llamadas individuales, los datos son procesados por los procedimientos y, una vez que el programa ha terminado, los datos resultantes son presentados.
Así, el flujo de datos puede ser ilustrado como una gráfica jerárquica, un árbol.

Programación modular.

En la programación modular, los procedimientos con una funcionalidad común son agrupados en módulos separados.
Un programa por consiguiente, ya no consiste solamente de una sección. Ahora está dividido en varias secciones más pequeñas que interactúan a través de llamadas a procedimientos y que integran el programa en su totalidad.

Cada módulo puede contener sus propios datos. Esto permite que cada módulo maneje un estado interno que es modificado por las llamadas a procedimientos de ese módulo.
Sin embargo, solamente hay un estado por módulo y cada módulo existe cuando más una vez en todo el programa.

Datos y Operaciones separados

La separación de datos y operaciones conduce usualmente a una estructura basada en las operaciones en lugar de en los datos : Los Módulos agrupan las operaciones comunes en forma conjunta.

Al programar entonces se usan estas operaciones proveyéndoles explícitamente los datos sobre los cuáles deben operar.

La estructura de módulo resultante está por lo tanto orientada a las operaciones más que sobre los datos. Se podría decir que las operaciones definidas especifican los datos que serán usados.

En la programación orientada a objetos, la estructura se organiza por los datos.

Se escogen las representaciones de datos que mejor se ajusten a tus requerimientos. Por consecuencia, los programas se estructuran por los datos más que por las operaciones.

Así, esto es exactamente del otro modo : Los datos especifican las operaciones válidas. Ahora, los módulos agrupan representaciones de datos en forma conjunta.

Programación orientada a objetos.

La programación orientada a objetos resuelve algunos de los problemas que se acaban de mencionar. De alguna forma se podría decir que obligan a prestar atención a los datos.

En contraste con las otras técnicas, ahora tenemos una telaraña de objetos interactuantes, cada uno de los cuáles manteniendo su propio estado.

En la programación orientada a objetos deberíamos tener tantos objetos de pila como sea necesario. En lugar de llamar un procedimiento al que le debemos proveer el manejador de la pila correcto, mandaríamos un mensaje directamente al objeto pila en cuestión.

En términos generales, cada objeto implementa su propio módulo, permitiendo por ejemplo que coexistan muchas pilas.

Cada objeto es responsable de inicializarse y destruirse en forma correcta. Por consiguiente, ya no existe la necesidad de llamar explícitamente al procedimiento de creación o de terminación.

¿No es ésta solamente una manera más elegante de técnica de programación modular ? Podría ser, si esto fuera todo acerca de la orientación a objetos. De hecho se puede tratar de programar de esta forma sin POO. Pero eso no es todo lo que es la POO.

Tipos de Datos Abstractos

Algunos autores describen la programación orientada a objetos como programación de tipos de datos abstractos y sus relaciones. Los tipos de datos abstractos son como un concepto básico de orientación a objetos.

Los Problemas

La primera cosa con la que uno se enfrenta cuando se escriben programas es el problema.

Típicamente, uno se enfrenta a problemas "de la vida real" y nos queremos facilitar la existencia por medio de un programa para dichos problemas.

Sin embargo, los problemas de la vida real son nebulosos y la primera cosa que se tiene que hacer es tratar de entender el problema para separar los detalles esenciales de los no esenciales : Tratando de obtener tu propia perspectiva abstracta, o modelo, del problema. Este proceso de modelado se llama abstracción y se ilustra en la Figura:

El modelo define una perspectiva abstracta del problema. Esto implica que el modelo se enfoca solamente en aspectos relacionados con el problema y que tú tratas de definir propiedades del problema. Estas propiedades incluyen

- los datos que son afectados
- las operaciones que son identificadas

por el problema.

Para resumir, la abstracción es la estructuración de un problema nebuloso en entidades bien definidas por medio de la definición de sus datos y operaciones. Consecuentemente, estas entidades combinan datos y operaciones. No están desacoplados unos de otras.

Tipos de Datos Abstractos y Orientación a Objetos

Los TDAs permiten la creación de instancias con propiedades bien definidas y comportamiento bien definido. En orientación a objetos, nos referimos a los TDAs como clases. Por lo tanto, una clase define las propiedades de objetos instancia en un ambiente orientado a objetos.

Los TDAs definen la funcionalidad al poner especial énfasis en los datos involucrados, su estructura, operaciones, así como en axiomas y precondiciones. Consecuentemente, la programación orientada a objetos es "programación con TDAs" : al combinar la funcionalidad de distintos TDAs para resolver un problema. Por lo tanto, instancias (objetos) de TDAs (clases) son creados dinámicamente, usados y destruídos.

Conceptos de básicos de objetos.

La programación tradicional separa los datos de las funciones, mientras que la programación orientada a objetos define un conjunto de objetos donde se combina de forma modular los datos con las funciones.

Aspectos principales:

1) Objetos.
- El objeto es la entidad básica del modelo orientado a objetos.
- El objeto integra una estructura de datos (atributos) y un comportamiento (operaciones).
- Se distinguen entre sí por medio de su propia identidad, aunque internamente los valores de sus atributos sean iguales.

2) Clasificación.
- Las clases describen posibles objetos, con una estructura y comportamiento común.
- Los objetos que contienen los mismos atributos y operaciones pertenecen a la misma clase.
- La estructura de clases integra las operaciones con los atributos a los cuales se aplican.

3) Instanciación.
- El proceso de crear objetos que pertenecen a una clase se denomina instanciación. (El objeto es la instancia de una clase).
- Pueden ser instanciados un número indefinido de objetos de cierta clase.

4) Generalización.

- En una jerarquía de clases, se comparten atributos y operaciones entre clases basados en la generalización de clases.
- La jerarquía de generalización se construye mediante la herencia.
- Las clases más generales se conocen como superclases. (clase padre)
- Las clases más especializadas se conocen como subclases (clases hijas).
- La herencia puede ser simple o múltiple.

5) Abstracción.
- La abstracción se concentra en lo primordial de una entidad y no en sus propiedades secundarias.
- Además en lo que el objeto hace y no en cómo lo hace.
- Se da énfasis a cuales son los objetos y no cómo son usados. *Logrando el desarrollo de sistemas más estables.*

6) Encapsulación.
- Encapsulación o encapsulamiento es la separación de las propiedades externas de un objeto de los detalles de implementación internos del objeto.
- Al separar la interfaz del objeto de su implementación, se limita la complejidad al mostrarse sólo la información relevante.
- Disminuye el impacto a cambios en la implementación, ya que los cambios a las propiedades internas del objeto no afectan su interacción externa.

7) Modularidad.

- El encapsulamiento de los objetos trae como consecuencia una gran modularidad.
- Cada módulo se concentra en una sola clase de objetos.
- Los módulos tienden a ser pequeños y concisos.
- La modularidad facilita encontrar y corregir problemas.
- La complejidad del sistema se reduce facilitando su mantenimiento.

8) Extensibilidad.
- La extensibilidad permite hacer cambios en el sistema sin afectar lo que ya existe.
- Nuevas clases pueden ser definidas sin tener que cambiar la interfaz del resto del sistema.
- La definición de los objetos existentes puede ser extendida sin necesidad de cambios más allá del propio objeto.

9) Polimorfismo.
- El polimorfismo es la característica de definir las mismas operaciones con diferente comportamiento en diferentes clases.
- Se permite llamar una operación sin preocuparse de cuál implementación es requerida en que clase, siendo responsabilidad de la jerarquía de clases y no del programador.

10) Reusabilidad de código.
- La orientación a objetos apoya el reuso de código en el sistema.
- Los componentes orientados a objetos se pueden utilizar para estructurar librerías resuables.
- El reuso reduce el tamaño del sistema durante la creación y ejecución.
- Al corresponder varios objetos a una misma clase, se guardan los atributos y operaciones una sola vez por clase, y no por cad objeto.
- La herencia es uno de los factores más importantes contribuyendo al incremento en el reuso de código dentro de un proyecto.

Abstracción de datos: Clases y objetos.

Clases.

Se mencionaba anteriormente que la base de la programación orientada a objetos es la abstracción de los datos o los TDAs. La abstracción de los datos se da realmente a través de las clases y objetos.

Def. Clase. Se puede decir que una clase es la implementación real de un TDA, proporcionando entonces la estructura de datos necesaria y sus operaciones. Los datos son llamados **atributos** y las operaciones se conocen como **métodos**. [6]

La unión de los atributos y los métodos dan forma al comportamiento de un grupo de objetos. La clase es entonces como la definición de un esquema dentro del cual encajan un conjunto de objetos.

Ejemplos de clases: automóvil, persona, libro, revista, reloj, silla,...

Objetos e instancias.

Una de las características más importantes de los lenguajes orientados a objetos es la instanciación. Esta es la capacidad que tienen los nuevos tipos de datos, para nuestro caso en particular las clases de ser "instanciadas" en cualquier momento.

El instanciar una clase produce un objeto o instancia de la clase requerida.

Def. Objeto. Un objeto es una instancia de una clase. Puede ser identificado en forma única por su nombre y define un estado, el cuál es representado por los valores de sus atributos en un momento en particular. [6]

El estado de un objeto cambia de acuerdo a los métodos que le son aplicados. Nos referimos a esta posible secuencia de cambios de estado como el comportamiento del objeto :

Def. Comportamiento. El comportamiento de un objeto es definido por un conjunto de métodos que le pueden ser aplicados.[6]

Instanciación.

Los objetos pueden ser creados de la misma forma que una estructura de datos:

1. Estáticamente. En tiempo de compilación se le asigna un área de memoria.
2. Dinámicamente. Se le asigna un área de memoria en tiempo de ejecución y su existencia es temporal. Es necesario liberar espacio cuando el objeto ya no es útil, para esto puede ser que el lenguaje proporcione mecanismos de recolección de basura.

Clases en C++.

Una clase entonces permite encapsular la información a través de atributos y métodos que utilizan la información, ocultando la información y la implementación del comportamiento de las clases.

La definición de una clase define nuevos TDAs y la definición en C++ consite de la palabra reservada class, seguida del nombre de la clase y finalmente el cuerpo de la clase encerrado entre llaves y finalizando con ;.

El cuerpo de la clase contiene la declaración de los atributos de la clase (variables) y la declaración de los métodos (funciones). Tanto los atributos como los métodos pertenecen exclusivamente a la clase y sólo pueden ser usados a través de un objeto de esa clase.

Sintaxis:

```
class <nombre_clase> {
        <cuerpo de la clase>
};
```

Ejemplo:

```
class cEjemplo1 {
        int x;
        float y;
        void fun(int a, float b) {
                x=a;
                y=b;
        }
};
```

Miembros de una clase.

Una clase está formada por un conjunto de miembros que pueden ser datos, funciones, clases anidadas, enumeraciones, tipos de dato, etc. (*amigos*). Por el momento nos vamos a centrar en los datos y las funciones (atributos y métodos.

Es importante señalar que un miembro no puede ser declarado más de una vez. Tampoco es posible añadir miembros después de la declaración de la clase.

Ejemplo:

```
class cEjemplo2{
        int i;
        int i;   //error
        int j;
        int func(int, int);
}
```

Atributos miembro.

Todos los atributos que forman parte de una clase deben ser declarados dentro de la misma.

Métodos miembro.

Los métodos al igual que los atributos, deber ser definidos en la clase, pero el cuerpo de la función puede ir dentro o fuera de la clase. Si un método se declara completo dentro de la clase, se considera como inline (puede ser en línea).

La declaración dentro de la clase no cambia con respecto a la declaración de una función, salvo que se hace dentro de la clase. Recordemos el ejemplo inicial pero ahora modificado con el cuerpo del méotdo fuera del cuerpo de la clase.

Ejemplo:

```
//código en ejemplo3.h
class cEjemplo3 {
        public:
        int x;
        float y;
        int funX(int a) {
                x=a;
                return x;
        }
        float funY(float);
};
```

Podemos ver que en la definición de la clase se incluye un método en línea y un prototipo de otro método.

Para definir un método miembro de una clase se debe escribir antes del nombre del método la clase con la que el método esta asociado. Para esto se ocupa el operador de resolución de alcance (o de ámbito) :: .

Continuación del ejemplo:

```
float cEjemplo3::funY(float b){
        y=b;
        return y;
}
```

Reiteramos que al declarar los métodos fuera de la clase no puede mencionarse la declaración de un método que no esté contemplado dentro de la clase, pues entonces cualquiera podría ganar acceso a la clase con sólo declarar una función adicional.

Ejemplo:

```
//error en declaración de un método
class x{
        public:
        int a;
        f();
};

int x::g() {        //error sólo se puede con f()
        return a*=3.1234;
}
```

La declaración de una función miembro es considerada dentro del ámbito de su clase. Lo cual significa que puede usar nombres de miembros de la clase directamente sin usar el operador de acceso de miembro de la clase.

Recordar que por convención en POO las funciones son llamadas métodos y la invocación o llamada se conoce como mensaje.

Un primer acercamiento al acceso a miembros.

Otra de las ventajas de la POO es la posibilidad de encapsular datos, ocultándolos de otros objetos si es necesario. Para esto existen principalmente dos calificadores que definen a los datos como públicos o privados.

Miembros públicos. Se utiliza cuando queremos dar a usuarios de una clase (*no implementadores)* el acceso a miembros de esa clase, los miembros deben ser declarados públicos.
Sintaxis.

```
public:
        <definición de miembros>
```

Miembros privados. Si queremos ocultar ciertos miembros de una clase de los usuarios de la misma, debemos declarar a los miembros como privados. De esta forma nadie más que los miembros de la clase pueden usar a los miembros privados. *Con excepción de las funciones amigas.* Por omisión los miembros se consideran privados. *En una estructura se consideran públicos por omisión.*
Sintaxis:

```
private:
        <definición de miembros>
```

Normalmente se trata de que los atributos de la clase sean privados, así como los métodos que no sean necesarios externamente o que puedan conducir a un estado inconsistente del objeto.[1]

En el caso de los atributos, estos al ser privados deberían de contar con métodos de modificación y de consulta pudiendo incluir alguna validación.

Es una buena costumbre de programación accesar a los atributos solamente a través de las funciones de modificación, sobre todo si es necesario algún tipo de verificación sobre el valor del atributo.

Ejemplo:

```
//código en ejemplo3.h
class cFecha {
        private:
        int dia;
        int mes;
        int an;

        public:
        char setDia(int);   //poner día
        int getDia();       //devuelve día
        char setMes(int);
        int getMes();
        char setAn(int);
        int getAnd();
};
```

Objetos de clase.

[1] Un estado inconsistente sería una modificación indebida de los datos, por ejemplo una modificación si validación.

Ya se ha visto como definir una clase, declarando sus atributos y sus operaciones, mismas que pueden ir dentro de la definición de la clase (inline) o fuera. Ahora vamos a ver como es posible crear objetos o instancias de esa clase.

Hay que recordar que una de las características de los objetos es que cada uno guarda un estado particular de acuerdo al valor de sus atributos[2].

Lo más importante de los LOO es precisamente el objeto, el cual es una identidad lógica que contiene datos y código que manipula esos datos. [5]

En C++, un objeto es una variable de un tipo definido por el usuario.[5]

Alcance de Clase.

El nombre de un miembro de una clase es local a la clase. Las funciones no miembros se definen en un alcance de archivo.

Dentro de la clase los miembros pueden ser accesados directamente por todos los métodos miembros. Fuera del alcance la clase, los miembros de la clase se pueden utilizar seguidos del operador de selección de miembro de punto . ó operador de selección de miembro de flecha ⇉ , posteriormente al nombre de un objeto de clase.

Ejemplo:

```
Class cMiClase{
        Public:
          Static int valor(); //no usar la declaración static todavía

          Int otraFuncion();
}

void main () {
        cMiClase cM;

        cM.otraFuncion();

        xMiClase::valor();
}
```

[2] A diferencia de la programación modular, donde cada módulo tiene un solo estado. Aunque está característica se puede lograr en la programación modular.

Sobrecarga de operaciones.

En C++ es posible tener el mismo nombre para una operación con la condición de que tenga parámetros diferentes. La diferencia debe de ser al menos en el tipo de datos. *Al menos un parámetro debe ser diferente.*

Si se tienen dos o más operaciones con el mismo nombre y diferentes parámetros se dice que dichas operaciones están sobrecargadas.

El compilador sabe que operación ejecutar a través de la firma de la operación, que es una combinación del nombre de la operación y el número y tipo de los parámetros.

El tipo de regreso de la operación puede ser igual o diferente.

La sobrecarga de operaciones sirve para hacer un código más legible y modular. La idea es utilizar el mismo nombre para operaciones relacionadas. Si no tienen nada que ver entonces es mejor utilizar un nombre distinto.

Ejemplo:

```
class cMiClase{
        int x;
        Public:
        void modifica() {
                X++;
        }
        void modifica(int y){
                x=y*y;
        }


}
```

Ejemplo 2:

```
//fuera de POO
#include <iostream.h>

int cuadrado(int i){
        return i*i;
}
double cuadrado(double d){
        return d*d;
}

void main() {
        cout<<"10 elevado al cuadrado:  "<<cuadrado(10)<<endl;
        cout<<"10.5 elevado al cuadrado: "<<cuadrado(10.5)<<endl;
}
```

Constructores y destructores.

Con el manejo de los tipos de datos primitivos, el compilador se encarga de reservar la memoria y de liberarla cuando estos datos salen de su ámbito.

En la programación orientada a objetos, se trata de proporcionar mecanismos similares. Cuando un objeto es creado es llamada un método conocido como constructor, y al salir se llama a otro conocido como destructor. Si no se proporcionan estos métodos se asume la acción más simple.

Constructor.

Un constructor es un método con el mismo nombre de la clase. Este método no puede tener un tipo de dato.

Ejemplo:

```
Class cCola{
 Private:
        Int q[100];
        Int sloc, rloc;
 Public:
        cCola( );//constructor
        Void put(int);
        Int get( );
};

//implementación del cosntructor
cCola::cCola ( ) {
        sloc=rloc=0;
        cout<<"Cola inicializada \n";
}
```

Destructor.

La contraparte del constructor es el destructor. Este se ejecuta momentos antes de que el objeto sea destruido, ya sea porque salen de su ámbito o por medio de una instrucción delete. El uso más común para un destructor es liberar la memoria asignada dinámicamente.

El constructor tiene al igual que el destructor el mismo nombre del constructor pero con una tilde como prefijo (~).

El destructor tampoco regresa valores ni tiene parámetros.

Ejemplo:

```
Class cCola{
 Private:
        Int q[100];
        Int sloc, rloc;
 Public:
        cCola( );//constructor
        ~cCola();  //destructor
        Void put(int);
```

```
        Int get( );
};

cCola::~cCola( ){
        cout<<"cola destruida\n";
}
```

Miembros estáticos.

Cada objeto tiene su propio estado, pero a veces es necesario tener valores por clase y no por objeto. En esos casos es necesario tener atributos estáticos que sean compartidos por todos los objetos de la clase.

Existe solo una copia de un miembro estático y no forma parte de los objetos de la clase.

ejemplo:

```
class cObjeto{
 private:
        char nombre[10];
        static int numObjetos;
 public:
        cObjeto(char *cadena=NULL);
        ~cObjeto();
};

cObjeto::cObjeto(char *cadena){
        if(cadena!=NULL)
                strcpy(nombre, cadena);
        else
                nombre=NULL;
        numObjetos++;
}
cObjeto::~cObjeto(){
        numObjetos--;
}
```

Un miembro estático es accesible desde cualquier objeto de la clase o mediante el operador de resolución de alcance binario (::) y el nombre de la clase, dado que existen aunque no haya instancias de la clase.

Sin embargo, el acceso sigue restringido bajo las reglas de acceso a miembros. Si se quiere accesar a un miembro estático que es privado deberá hacerse mediante un método público. Si no existe ningunaq instancia de la clase entonces deberá ser por medio de un método público y estático.

Un método estático solo puede tener acceso a miembros estáticos.

Los atributos estáticos deben de ser inicializados al igual que los atributos constantes, fuera de la declaración de la clase. Ejemplo:

```
int cClase::atributo=0;          int const cClase::ATRCONST=50;
```

Objetos constantes.

Es posible tener objetos de tipo constante, los cuales no podrán ser modificados en ningún momento.[3] Tratar de modificar un objeto constante se detecta como un error en tiempo de compilación.

Sintaxis:

```
const <clase> <lista de objetos>;        const cHora h1(9, 30, 20);
```

Para estos objetos, algunos compiladores llegan a ser tan rígidos en el cumplimiento de la instrucción, que no permiten que se hagan llamadas a métodos sobre esos objetos. En el caso de C++ de Borland, el compilador únicamente manda una advertencia y permite que se ejecute, pero advierte que debe ser considerado como un error.

Sin embargo, es posible que se quiera consultar al objeto mediante llamadas a métodos get, para esto se pueden declarar métodos con la palabra reservada const, para permitirles actuar libremente sobre los objetos sin modificarlo. La sintaxis es añadir después de la lista de parámetros la palabra reservada conts en la declaración y en su definición.

Sintaxis:
Declaración:

```
<tipo> <nombre> (<parámetros>) const;
```

Definición del método fuera de la declaración de la clase:

```
<tipo> <clase>::<nombre> (<parámetros>) const {
        <código>
}
```

Definición del método dentro de la declaración de la clase:

```
<tipo> <nombre> (<parámetros>) const {
        <código>
}
```

Aunque el compilador de borland permite usar los métodos constantes de manera indiferente para objetos constantes y no constantes siempre y cuando no modifiquen al objeto; sin embargo, algunos compiladores restringen el uso de métodos constantes a objetos constantes. Para solucionarlo es posible sobrecargar el método con la única diferencia de la palabra const, aunque el resto de la firma del método sea la misma.

Los constructores no necesitan la declaración const, puesto que deben poder modificar al objeto.

[3] Ayuda a cumplir el principio del mínimo privilegio, donde se debe restringir al máximo el acceso a los datos cuando este acceso estaría de sobra. [1]

Objetos compuestos.

Algunas veces una clase no puede modelar adecuadamente una entidad basándose únicamente en tipos de datos simples. Los LPOO permiten a una clase contener objetos. Un objeto forma parte directamente de la clase en la que se encuentra declarado. El objeto compuesto es una especie de relación, pero con una asociación más fuerte con los objetos relacionados. A la noción de objeto compuesto se le conoce también como objeto complejo o agregado. (términos similares: objetos compuestos, composición, agregación, objeto complejo

Rumbaugh en [10] define a la agregación como "una forma fuerte de asociación, en la cual el objeto agregado está formado por componentes. Los componentes forman parte del agregado. El agregado, es un objeto extendido que se trata como una unidad en muchas operaciones, aun cuando conste físicamente de varios objetos menores."

Ejemplo: Un automóvil se puede considerar ensamblado o agregado, donde el motor y la carrocería serian sus componentes.

- Pedir ejemplos a los alumnos. (bici, computadora,...)

El concepto de agregación puede ser relativo a la conceptualización que se tenga de los objetos que se quieran modelar. Hay que tener en cuenta que pasa con los objetos que son parte del objeto compuesto cuando éste último se destruye. En general hay dos opciones:

1. Cuando el objeto agregado se destruye, los objetos que lo componen no tienen necesariamente que ser destruidos.
2. Cuando el agregado es destruido también sus componentes se destruyen.

Por el momento vamos a considerar la segunda opción, por ser más fácil de implementar y resulta de los objetos que se encuentran embebidos como un atributo más una clase.

Ejemplo:

```
class Nombre {
private:
      char paterno[20],
        materno[20],
        nom[15];
public:
      set(char,*, char*, char *);
      ...
};

class Persona {
private:
      int edad;
      Nombre nombrePersona;
....
};
```

Al crea un objeto compuesto, cada uno de sus componentes es creado con sus respectivos constructores. Para inicializar esos objetos componentes tenemos dos opciones:

1. En el constructor del objeto compuesto llamar a los métodos set correspondientes a la modificación de los atributos de los objetos componentes.
2. Pasar en el constructor del objeto compuesto los argumentos a los constructores de los objetos componentes.

Sintaxis:

<clase>::<constructor>(<lista de argumentos>) : <objeto componente 1>(<lista de argumentos sin el tipo>),...

donde la lista de argumentos del objeto compuesto debe incluir a los argumentos de los objetos componentes, para que puedan ser pasados en la creación del objeto.

Un objeto que es parte de otro objeto, puede a su vez ser un objeto compuesto. De esta forma podemos tener múltiples niveles . Un objeto puede ser un agregado recursivo, es decir, tener un objeto de su misma clase. Ejemplo: Directorio de archivos.

Asociaciones entre clases.

Una clase puede estar relacionada con otra clase, o en la práctica un objeto con otro objeto. En el modelado de objetos a la relación entre clases se le conoce como asociación; mientras que a la relación entre objetos se le conoce como liga. Por lo tanto, una liga es una instancia de una asociación.

Ejemplo:
Una clase estudiante está relacionada con una clase Universidad.

Una relación es una conexión física o conceptual entre objetos. Las relaciones se consideran de naturaleza bidireccional; es decir, ambos lados de la asociación tienen acceso a clase del otro lado de la asociación. Sin embargo, algunas veces únicamente es necesario una relación en una dirección.

Comúnmente las asociaciones se representan en los LPOO como apuntadores. Donde un apuntador a una clase B en una clase A indicaría la asociación que tiene A con B; aunque no así la asociación de B con A.
Para una asociación bidireccional es necesario al menos un par de apuntadores, uno en cada clase. Para una asociación undireccional basta un solo apuntador en la clase que mantiene la referencia.

Si una clase mantiene una asociación consigo misma se dice que es una asociación reflexiva. Ejemplo: Persona puede tener relaciones entre si, si lo que nos interesa es representar a las personas que guardan una relación entre sí, por ejemplo si son parientes. Es decir, un objeto mantiene una relación con otro objeto de la misma clase.

Ejemplo: un programa que guarda una relación bidireccional entre clases A y B.

```
Class A{
        //lista de atributos

        B       *pB;
}

class B{
        //lista de atributos
        A       *pA;
}
```

En el ejemplo anterior se presenta una relación bidireccional, por lo que cada clase tiene su respectivo apuntador a la clase contraria de la relación. Además, deben proporcionarse métodos de acceso a la clase relacionada por medio del apuntador.

En el caso de las relaciones se asumirá que cada objeto puede seguir existiendo de manera independiente, a menos que haya sido creada por la instancia de la clase relacionada, en cuyo caso deberá ser eliminada por el destructor del objeto que la creo. Es decir, si el objeto A crea al objeto B, es responsabilidad de A eliminar a la clase B antes de que A sea eliminada. En caso contrario, si B es independiente de la clase A, A debería enviar un mensaje al objeto B para que asigne NULL al apuntador de B o para que tome una medida pertinente, de manera que no quede apuntando a una dirección inválida. Es importante señalar que las mnedidad que se tomen pueden variar de acuerdo a las necesidades de la aplicación, pero bajo ningún motivo se deben dejar accesos a áreas de memoria no permitidas o dejar objetos "volando", sin que nadie haga referencia a ellos.

Tenemos a continuación dos ejemplos de estructuras clásicas.:

1. Ejemplo de relación unidireccional: lista ligada
2. Ejemplo de relación bidireccional: lista doblemente ligada.

Multiplicidad de una asociación.

La multiplicidad de una asociación especifica cuantas instancias de una clase se pueden relacionar a **una sola instancia** de otra clase.

Se debe determinar la multiplicidad para cada clase en una asociación.

Tipos de asociaciones según su multiplicidad:

ß "uno a uno": donde dos objetos se relacionan de forma exclusiva, uno con el otro.
Ejemplo: Un alumno tiene una boleta de calificaciones.

ß "uno a muchos": donde uno de los objetos puede estar relacionado con muchos otros objetos.
Ejemplo: un usuario de la biblioteca puede tener muchos libros prestados.

ß "muchos a muchos": donde cada objeto de cada clase puede estar relacionado con muchos otros objetos.
Ejemplo: Un libro puede tener varios autores, mientras que un autor puede tener varios libros.

La forma de implementar en C++ este tipo de relaciones puede variar, pero la más común es por medio de apuntadores a objetos: *(considere relaciones bidireccionales)*

ß "uno a uno". Un apuntador de cada lado de la relación, como se ha visto anteriormente.
ß "uno a muchos". Un apuntador de un lado y un arreglo de apuntadores a objetos definido dinámica o estáticamente.

```
Class A{
        ...
        B *pB;
};
class B{
        A *p[5];
    //ó
        A **p;
}
```

Otra forma es manejar una clase que agrupe a pares de direcciones en un objeto independiente de la clase. Por ejemplo una lista.

ß "muchos a muchos". Normalmente se utiliza un objeto u objetos independientes que mantiene las relaciones entre los objetos.

Constructor de Copia

Es útil agregar a todas las clases un constructor de copia que reciba como parámetro un objeto de la clase y copie sus datos al nuevo objeto. C++ proporciona un constructor de copia por omisión, sin embargo es una copia a nivel de miembro y puede no realizar una copia exacta de lo que queremos. Por ejemplo en casos de apuntadores a memoria dinámica, se tendría una copia de la dirección y no de la información referenciada. Sintaxis clásica:

<nombre clase>(const <nombre clase> &<objeto>)...

Sobrecarga de operadores.

C++ no permite la creación de nuevos operadores, pero si permite en cambio sobrecargar los operadores existente para que se utilicen con los objetos. De esta forma se les da a los operadores un nuevo significado de acuerdo al objeto sobre el cual se aplique.

Para sobrecargar un operador, se define un método que es invocado cuando el operador es aplicado sobre ciertos tipos de datos.

Para utilizar un operador con objetos, es necesario que el operador este sobrecargado, aunque existen dos excepciones:

- El operador de asignación =, puede ser utilizado sin sobrecargarse explícitamente, pues el comportamiento por omisión es una *copia a nivel de miembro* de los miembros de la clase. Sin embargo no debe de usarse si la clase cuenta con miembros a los que se les asigne memoria de manera dinámica.
- El operador de dirección &, esta sobrecargado por omisión para devolver la dirección de un objeto de cualquier clase.

Algunas restricciones:

1. Operadores que no pueden ser sobrecargados:

   . .* :: ?: sizeof

2. La precedencia de un operador no puede ser modificada. *Utilizar los paréntesis para obligar un nuevo orden de evaluación.*
3. La asociatividad de un operador no puede ser modificada.
4. No se puede modificar el número de operandos de un operador. Los operadores siguen siendo unarios o binarios.
5. No es posible crear nuevos operadores.
6. No puede modificarse el comportamiento de un operador sobre tipos de datos definidos por el lenguaje.

La sintaxis para definir un método con un operador difiere de la definición normal de un método, pues debe indicarse el operador seguido de la palabra reservada *operator* :

<tipo> *operator* <operador> (<argumentos>) ;
ó

<tipo> *operator* <operador> (<argumentos>) {
	<cuerpo del método>
}

Para la definición fuera de la clase:
<tipo> <clase>::*operator* <operador> (<argumentos>) {
	<cuerpo del método>
}

Funciones amigas (friends).

En Objetos existe la amistad. Aunque a algunos la consideran como una intrusión a la encapsulación o a la privacidad de los datos: *"... la amistad corrompe el ocultamiento de información y debilita el valor del enfoque de diseño orientado a objetos"[1].*

Un amigo de una clase es una función u otra clase que no es miembro de la clase, pero que tiene permiso de usar los miembros públicos y privados de la clase.

El ámbito de una función amiga no es el de la clase, y los amigos no son llamados con los operadores de acceso de miembros.

Sintaxis para una función amiga:

```
Class c1{
	friend void miamiga(int);
	...
public:
	...
};
```

Sintaxis para una clase amiga:

```
Class c1{
	friend c2;
	...
public:
	...
};
```

Las funciones o clases amigas no son privadas ni públicas (o protegidas), pueden ser colocadas en cualquier parte de la definición de la clase, pero se acostumbra que sea al principio.[1]

Como la amistad entre personas, esta es concedida y no tomada. Si la clase B quiere ser amigo de la clase A, la clase A debe declarar que la clase B es su amigo.

La amistad no es simétrica ni transitiva: si la clase A es un amigo de la clase B, y la clase B es un amigo de la clase C, no implica:

1. Que la clase B sea un amigo de la clase A.
2. Que la clase C sea un amigo de la clase B.
3. Que la clase A sea un amigo de la clase C.

Herencia.

La herencia es un mecanismo potente de abstracción que permite compartir similitudes entre clases manteniendo al mismo tiempo sus diferencias.

Es una forma de reutilización de código, tomando clases previamente creadas y formando a partir de ellas nuevas clases, heredándoles sus atributos y métodos. Las nuevas clases pueden ser modificadas agregándoles nuevas características.

En C++ la clase de la cual se toman sus características se conoce como clase base; mientras que la clase que ha sido creada a partir de la clase base se conoce como clase derivada. Existen otros términos para estas clases:

Clase base	Clase derivada
Superclase	Subclase
Clase padre	Clase hija

Una clase derivada es potencialmente una clase base, en caso de ser necesario.

Cada objeto de una clase derivada también es un objeto de la clase base. En cambio, un objeto de la clase base no es un objeto de la clase derivada.

La implementación de herencia a varios niveles forma un árbol jerárquico similar al de un árbol genealógico. Este es conocida como jerarquía de herencia.

Generalización. Una clase base o superclase se dice que es más general que la clase derivada o subclase.

Especialización. Una clase derivada es por naturaleza una clase más especializada que su clase base.

Implementación en C++.

La herencia en C++ es implementada permitiendo a una clase incorporar a otra clase dentro de su declaración.

Sintaxis general:

```
Class class_nueva: acceso  clase_base {
        //cuerpo clase nueva
};
```

Control de Acceso a miembros.

Existen tres palabras reservadas para el control de acceso: *public, private y protected*. Estas sirven para proteger los miembros de la clase en diferentes formas. El control de acceso, como ya se vio anteriormente, se aplica a los métodos, atributos, constantes y tipos anidados que son miembros de la clase.

Resumen de tipos de acceso:

Tipo de acceso	Descripción
Private	Un miembro privado únicamente puede ser utilizado por los métodos miembro y funciones amigas de la clase donde fue declarado.
Protected	Un miembro protegido puede ser utilizado únicamente por los métodos miembro y funciones amigas de la clase donde fue declarado o por los métodos miembro y funciones amigas de las clases derivadas.

	El acceso protegido es como un nivel intermedio entre el acceso privado y público.
Public	Un miembro público puede ser utilizado por cualquier método. Una estructura es considerada por C++ como una clase que tiene todos sus miembros públicos.

Control de acceso en herencia.

Hasta ahora se han usado la herencia con un solo tipo de acceso, usando el especificador *public*, los miembros públicos de la clase base son miembros públicos de la clase derivada, los miembros protegidos permanecen protegidos para la clase derivada.

Para accesar a los miembros de una clase base desde una clase derivada, se pueden ajustar los permisos por medio de un calificador *public*, *private* o *protected*.

Si una clase base es declarada como pública de una clase derivada, los miembros públicos y protegidos son accesibles desde la clase derivada, no así los miembros privados.

Si una clase base es declarada como privada de otra clase derivada, los miembros públicos y protegidos de la clase base serán miembros privados de la clase derivada. Los miembros privados de la clase base permanecen inaccesibles.

Si se omite el calificador de acceso de una clase base, se asume por default que el calificador es *public* en el caso de una estructura y *private* en el caso de una clase.

Ejemplo de sintaxis:

```
Class base {
 ...
};

class d1: private base {
...
};
class d2: base {
...
};

class d3: public base {
...
};
```

Es recomendable declarar explícitamente la palabra reservada private al tomar una clase base como privada para evitar confusiones:

```
Class x{
 Public:
      F();
};

class y: x {   //privado por omisión
...
};
```

```
void g( y *p){
        p->f();  //error
}
```

Si una clase base es declarada como protegida de una clase derivada, los miembros públicos y protegidos de la clase base, se convietern en miembros protegidos de la clase derivada.

Manejo de objetos de la clase base como objetos de una clase derivada y viceversa.

Un objeto de una clase derivada pública, puede ser manejado como un objeto de su clase base. Sin embargo, un objeto de la clase base no es posible tratarlo de forma automática como un objeto de clase derivada.

Si se realiza la conversión explícita de un apuntador de clase base - que apunta a un objeto de clase base- a un apuntador de clase derivada y posteriormente, se hace referencia a miembros de la clase derivada, es un error pues esos miembros no existen en el objeto de la clase base.

Constructores de clase base

El constructor de la clase base puede ser llamado desde la clase derivada, para inicializar los atributos heredados. La sintaxis es igual que el inicializador de objetos componentes.

Los constructores y operadores de asignación de la clase base no son heredados por las clases derivadas. Pero pueden ser llamados por los de la clase derivada.

Un constructor de la clase derivada llama primero al constructor de la clase base. Si se omite el constructor de la clase derivada, el constructor por omisión de la clase derivada llamará al constructor de la clase base.

Los destructores son llamados en orden inverso a las llamadas del constructor: uncosntructor de una clase derivada será llamado antes que el de su clase base.

Redefinición de métodos

Algunas veces, los métodos heredados ya no cumplen completamente la función que quisiéramos que realicen. Es posible en C++ redefinir un método de la clase base en la clase derivada. Cuando se hace referencia al nombre del método, se ejecuta le versión de la clase en donde fue redefinida. Es posible sin embaro utilizar el método de la clase base por medio del operador de resolución de alcance.

De hecho se sugiere redefinir métodos que no vayan a ser empleados en la clase derivada, inclusive sin código para inhibir cualquier acción que no nos interese.

Herencia Múltiple.

Una clase puede heredar miembros de dos o más clases; lo que se conoce como herencia múltiple. Herencia múltiple es entonces, la capacidad de una clase derivada de heredar miembros de varias clases base.

Sintaxis:

```
Class <nombre clase derivada> : <clase base 1> , <clase base 2>, ...<clase base n> {
        ...
};
```

Ejemplo:

```
Class A{
Public:
        Int i;
        Int a();
};
class B{
public:
        int j;
        int b();
};
class C{
public:
        int k;
        int c();
};
class D: public A, public B, public C {
public:
        int l;
        int d();
};

void main() {
        D var1;

        Var1.a();
        Var1.b();
        Var1.c();
        Var1.d();
}
```

Puede tenerse un problema de ambigüedad al heredar dos métodos con el mismo nombre de clases diferentes. Se resuelve poniendo antes del nombre del miembro el nombre de la clase: *objeto.<clase::>miembro.* El nombre del objeto es necesario pues no se esta haciendo referencia a un miembro estático.

Constructores.

Si los constructores de la clase base no tienen argumentos, no es necesario crear un constructor de la clase derivada. Pero si hay constructores con argumentos, es necesario llamarlos desde el constructor de la clase derivada. Para ejecutar los constructores de las clases base, pasando los argumentos, es necesario especificarlos después de la declaración de la función de construcción de la clase derivada, separados por coma. Sintaxis general:

<Constructor clase derivada>(<argumentos> : <base1>(<argumentos>), <base2> (<argumentos>), ... , <basen>(<argumentos>) {
    ...
}

donde como en la herencia simple, el nombre base corresponde al nombre de la clase, o en este caso, clases base.

Funciones virtuales y polimorfismo.

La capacidad de polimorfismo permite crear programas con mayores posibilidad de expansiones futuras, aún para procesar en cierta forma objetos de clases que no han sido creadas o están en desarrollo [1].

El polimorfismo es implementado en C++ a través de clases derivadas y funciones virtuales.

Una función virtual es un método miembro declarado como *virtual* en una clase base y siendo este método redefinido en una o más clases derivadas.

Las funciones virtuales son muy especiales, debido a que cuando una función es accesada por un apuntador a una clase base, y este esta haciendo referencia a un objeto de una clase derivada, el programa determina en tiempo de ejecución a que función llamar, de acuerdo al tipo de objeto al que se apunta. Esto se conoce como ligadura tardía y el compilador de C++ incluye en el código máquina el manejo de ese tipo de asociación de métodos.

La utilidad se da cuando se tiene una método en una clase base, y este es declarado virtual. De esta forma, cada clase derivada puede tener su propia implementación del método si es que así lo requiere la clase, y si un apuntador a clase base hace referencia a cualquiera de los objetos de clases derivadas, el método que se ejecuta se determina dinámicamente cual de todos los métodos debe ejecutar.

La sintaxis en C++ implica declarar al método de la clase base con la palabra reservada *virtual*, redefiniendo ese método en cada una de las clases derivadas.

Al declarar un método como virtual, este método se conserva asi a través de toda la jerarquía de herencia, del punto en que se declaro hacia abajo. Aunque de este modo no es necesario volver a usar la palabra virtual en ninguno de los métodos inferiores del mismo nombre, es posible hacerlo de forma explícita para que el programa sea más claro.

Es importante señalar que las funciones virtuales que sean redefinidas en clases derivadas deben tener además de la misma firma que la función virtual base, el mismo tipo de retorno.

Sintaxis:
Class base {
    Virtual <tipo> <método> (<parámetros);
};

Hay que hacer notar que las funciones virtuales puede seguirse usando sin apuntadores, mediante un objeto de la clase. De esta forma, el método a ejecutar se determina de manera estática; es decir, en tiempo de compilación (ligadura estática). Obviamente el método a ejecutar es aquel definido en la clase del objeto o el heredado de su clase base.

Si se declara en una clase derivada un método con otro tipo de dato como retorno el compilador manda un error, ya que esto no es permitido. Si se declara un método con el mismo nombre pero diferentes parámetros, la función virtual queda desactivada de ese punto hacia abajo en la jerarquía de herencia.

Clase abstracta y clase concreta.

Existen clases que son útiles para representar una estructura en particular, pero que no van a tener la necesidad de generar objetos directamente a partir de esa clase, estas se conocen como clases abtractas, o mejor dicho como clases base abstractas, puesto que sirven para definir una estructura jerarquica.

La clase base abstracta entonces, tiene como objetivo proporcionar una clase base que ayude al modelado de la jerarquía de herencia, aunque esta sea muy general y no sea práctico tener instancias de esa clase. Por lo tanto de una clase abstracta no se pueden definir objetos, mientras que en clases a partir de las cuales se puedan instancias objetos se conocen como clases concretas.

En C++, una clase se hace abstracta al declarar al menos uno de los métodos como virtuales como puro. Un método o función virtual pura es una que en su declaración tiene el inicializador de =0 :

Virtual <tipo> <nombre>(<parámetros>) =0; //virtual pura

Es importante tener en cuenta que una clase sigue siendo abstracta hasta que no se implemente la función virtual pura, en una de las clases derivadas. Si no se hace la implementación, la función se hereda como virtual pura y por lo tanto la clase sigue siendo considerada como abstracta.

Aunque no se pueden tener objetos de clases abstractas, si se pueden tener apuntadores a objetos de esas clases, permitiendo una manipulación de objetos de las clases derivadas mediante los apuntadores a la clase abstracta.

Polimorfismo.

El polimorfismo se define como la capacidad de objetos de clases diferentes, relacionados mediante herencia, a responder de forma distinta a una misma llamada de un método. [1]

En C++, el polimorfismo se implementa con las funciones virtuales. Al hacer una solicitud de un método, a través de un apuntador a clase base para usar un método virtual, C++ determina el método que corresponda al objeto de la clase a la que pertenece, y no el método de la clase base.

Tener en cuenta que no es lo mismo que simplemente redefinir un método de clase base en una clase derivada, pues como se vio anteriormente, si se tiene a un apuntador de clase base y a través de el se hace la llamada a un método, se ejecuta el método de la clase base independientemente del objeto referenciado por el apuntador. Este no es un comportamiento polimórfico.

Destructores virtuales.

Cuando se aplica la instrucción *delete* a un apuntador de clase base, será ejecutado el destructor de la clase base sobre el objeto, independientemente de la clase a la que pertenezca. La solución es declarar al

destructor de la clase base como virtual. De esta forma al borrar a un objeto se ejecutará el destructor de la clase a la que pertenezca el objeto referenciado, a pesar de que los destructores no tengan el mismo nombre.

Un constructor no puede ser declarado como virtual.

Bibliografía

1. Deitel, P.J.; Deitel H.M. **COMO PROGRAMAR EN C/C++**. Prentice Hall. 2ª edición. México. 1995.
2. Schildt, Herbert. **APLIQUE TURBO C++.** Osbrone/ Mc Graw-Hill. 1ª edición. 1991.
3. Cox, Brad j.; Novobilski, Andrew J. **PROGRAMACIÓN ORIENTADA A OBJETOS. Un enfoque evolutivo**. Addison Wesley/Díaz de Santos. 2ª edición. 1993.
4. Pappas, chris H.; Murray, William H. **MANUAL DE BORLAND C++.** Mc Graw-Hill. 3ª edición. 1993.
5. Fundación Arturo Rosenblueth. **PROGRAMACIÓN ORIENTADA A OBJETOS EN C++.** México.
6. Müller, Peter. **INTRODUCCIÓN A LA PROGRAMACIÓN ORIENTADA A OBJETOS EMPLEANDO C++.** Globewide Network Academy (GNA). 1997. http://uu-gna.mit.edu:8001/uu-gna/text/cc/Tutorial//tutorial.html
7. Weitzenfeld, Alfredp. **PARADIGMA ORIENTADO A OBJETOS.** División Académica de Computación. Depto. Académico de Computación. ITAM. México. 1994.

---

*Notas en revisión.